AD-A213 955

WORKING MATERIAL

FOR THE LECTURES OF

Wilfried Brauer

FORMAL APPROACHES TO CONCURRENCY

INTERNATIONAL SUMMER SCHOOL

ON

LOGIC, ALGEBRA AND COMPUTATION

MARKTOBERDORF, GERMANY, JULY 25 - AUGUST 6,1989

THIS SUMMER SCHOOL IS ORGANIZED UNDER THE AUSPICES OF THE TECHNISCHE UNIVERSITÄT MÜNCHEN AND IS SPONSORED BY THE NATO SCIENCE COMMITEE AS PART OF THE 1989 ADVANCED STUDY INSTITUTES PROGRAMME. PARTIAL SUPPORT FOR THE CONFERENCE WAS PROVIDED BY THE EUROPEAN RESEARCH OFFICE AND THE NATIONAL SCIENCE FOUNDATION AND BY VARIOUS INDUSTRIAL COMPANIES.

Formal Approaches to Concurrency

Wilfried Brauer

Institut für Informatik
Technische Universität München
Arcisstr. 21, D-8000 München 2, FRG

0 Introduction

Formal (algebraic, combinatorial, logical) treatment of concurrent processes and of distributed systems has started rather recently only, although concurrent and distributed activities dedicated to common tasks are daily practice and have always played an important role for human societies. The traditional notions of computability are all based on the concept of a single person fulfilling a task step by step. This is very explicit in Turing's work – and even if the formalism would allow for consideration of concurrency, as in the case of recursive functions defined by sets of equations, this possibility was not discussed for a long time.

Nevertheless there are connections of the current theories of concurrency to the different approaches to formalize the notion of effective computation: Turing machines, λ-calculus and recursive functions. The two main approaches to concurrency that will be described in the following, namely Petri nets and abstract programming languages, are closely related to them. A Petri net can be understood as a formalization of the joint work of a group of people (see e.g. [Bra84] and [Bra87]), the *abstract programming* languages are greatly influenced by the ideas around the λ-calculus (see e.g. [Tra88]), their purpose is to prescribe what should be done by cooperating agents.

The rather recent intensification and broadening of work on concurrency is certainly due to hardware developments – but the development of theoretical informatics is also based on its own inherent impetus, in particular

on historical influences and on abstract (not hardware-oriented) ideas and concepts (which often can be developed by looking at what human beings do).

The two approaches we will deal with are quite different, from the formal and technical point of view as well as from the philosophical one. However, I will not concentrate on their distinctions, but treat them together from the perspective of specification and programming. Since in the case of distributed systems there is no clear distinction between specification and programming notations I shall use often the more general term "specification" to mean also programming.

The following five parts (corresponding to five lectures) are mainly based on work done in my research group, in particular by Astrid Kiehn, Dirk Taubner and Walter Vogler.

1 Abstract Programming Languages

1.1 A General Abstract Programming Language

Let us imagine that we should specify a distributed system composed of several agents which work rather independently but which communicate with each other (in a well organized way). To make the problem easier, we abstract from the processing of data and take as a basis simply a countably infinite alphabet

$$Alph$$

of (names of) **actions** (assuming also that the occurrence of an action in the specification of a system means that, in the realization of the system there will be an agent performing this action).

Naturally one would like to be able to describe simple systems like finite non-deterministic automata – however, we do not want to describe their structure, but their behaviour, i.e. what they should do. Therefore we use a notation similar to that of regular expressions; the main difference is that we will express the iteration (Kleene star) by **recursion**.

In addition we obviously need an operator for some sort of **parallel composition** which should include the possibility to prescribe communications or joint actions of the composed systems. There are several operators in the literature based on different ways of cooperation: the two subsystems may

- operate completely independently
- perform some actions jointly
- communicate by performing complementary actions a and $\overline{a}$ (establishing a communication link) – the joint action $(a, \bar{a})$ having no effect to the outside world (the communictation is internal).

We will use an operator which encompasses all these variants.

Obviously we now need a complementary alphabet $\overline{Alph} := \{\bar{a} \mid a \in A\}$ and a notation (τ) for an action without any (visible) effect. Naturally we assume that $\bar{\bar{a}} = a$.

When we specify a system we take the point of view of an observer (or a user) who watches (or interacts with) the system and sees the effects of its actions, i.e. of actions from

$$Vis := Alph \cup \overline{Alph}.$$

According to good programming practise we would also like to be able to express hiding (abstraction) and renaming of actions. Both can be combined in the operation of applying an **action manipulation** function f to a specification (It is convenient to write this operator in postfix notation.). So $af = \tau$ denotes that a is hided. We can use this operator also to disallow (restrict) actions, if we extend its range by $\bot$, the symbol for non-action, undefinedness etc. (i.e. $af = \bot$ means action a is not allowed).

It is convenient to have a notation for unordered pairs (of jointly executed) actions:

$$EVis := \{\{a, b\} \mid a, b \in Vis\} \qquad (\{a, a\} = \{a\})$$

The set of all actions is $Act := \{\tau\} \cup Vis \cup EVis$. Let moreover $Act_\bot := Act \cup \{\bot\}$. We are now ready to define the syntax of the **general abstract programming language GAP** (i.e. the language A in [Tau88]).

The **operators** (and their intuitive meanings) are:

nil: ` nullary operator (a system which is unable to perform any action; which has stopped to work)

a:	unary operator,	used in prefix notation for each $a \in \{\tau\} \cup Vis$ (called *prefixing*; if S is a system aS is the system that can perform first a and then behaves as S)
f:	unary operator,	used in postfix notation for each $f \in Fun := \{f : Act_\perp \to Act_\perp \mid f(\perp) = \perp, f(\tau) = \tau\}$ (called *action manipulation*)
$+$:	binary operator,	used in infix notation (called *sum*; $S + S'$ behaves either like S or like S', depending on whether the first executed action belongs to S or to S')
$\sharp$:	binary operator,	used in infix notation (called *general parallel composition*; $S \sharp S'$ allows S and S' to work independently but also to perform joint actions $\{a, b\} \in EVis$ provided that S, S' can perform a, b respectively.

The notation for recursion is $\mathrm{rec}\, r.S$, where $r \in Id$, a countably infinite set of identifiers, and S a system description in which r might occur. (This is similar to the definition of a parameterless recursive procedure r with procedure body S together with an immediate call of r.)

As usual we have the notions of free and bound identifiers, we have to use renaming of bound identifiers, we identify terms which differ only with respect to bound identifiers, and we will always assume that the Barendregt convention is obeyed, i.e. that in each collection of terms no identifier occurring bound in one of the terms occurs also free in a term of this collection.

Now the **syntax** for GAP is given by the grammar:

$$S ::= nil \mid r \mid aS \mid Sf \mid S + S \mid S \sharp S \mid \mathrm{rec}\, r.S$$

where $r \in Id$, $a \in \{\tau\} \cup Vis$, $f \in Fun$. Let $Term_{GAP}$ be the set of all terms defined by this grammar.

1.2 Derived Operators

Many of the operators used in the literature can be defined with the help of the above; here are some examples. Let $S, S' \in Term_{GAP}$, $A \subseteq Vis$, then:

S **or** S' := $\tau S + \tau S'$ is the internal nondeterminism operator of TCSP (without visible effect the system decides to behave like S or like S')

$S \setminus A$:= $S\{a \mapsto \tau \mid a \in A\}$ is the hiding operator of TCSP (Here as in the following we describe a function by writing down all important argument-value pairs)

$S - A$:= $S\{a \mapsto \bot \mid a \in A\}$ is the restriction operator of CCS

$S \mid S'$:= $(S \nmid S')g$, where $ag := \begin{cases} a & \text{for } a \in Vis \cup \{\tau\} \\ \tau & \text{for } a = \{b, \bar{b}\} \in EVis, \\ \bot & \text{otherwise} \end{cases}$
is the CCS parallel composition.

$S \parallel_A S'$:= $(S \nmid S')g_A$,
where $ag_A := \begin{cases} a & \text{for } a \in \{\tau\} \cup Vis - A \\ b & \text{for } a = \{b, b\} \in EVis,\ b \in A, \\ \bot & \text{otherwise} \end{cases}$
is the TCSP parallel composition.

Milner's pure CCS, the perhaps most influential abstract programming language, developed from the middle of the 70's on, (see [Mil85] and [BRR87]) is basically given by the following grammar

$$S ::= nil \mid r \mid aS \mid Sf \mid S - A \mid S + S \mid S \mid S \mid \mathrm{rec}\, r.S$$

where $r \in Id$, $a \in \{\tau\} \cup Vis$,

$f \in Fun$, such that $f\lceil_{EVis} = id \wedge \forall a \in Vis : af \in Vis \wedge \overline{af} = \bar{a}f$.

$A \subseteq Vis$, such that $a \in A$ implies $\bar{a} \in A$. (where $g\lceil_D$ denotes the restriction of the domain of the function g to D)

The classical operator ";" of sequential composition of two systems is not simply obtained from the prefixing operator, since we have allowed the construction of systems, which may never terminate their activities. We therefore introduce a particular symbol $\surd$ (called tick) which indicates successful termination. Let $Alph = Alph' \cup \{\surd, \surd_1, \surd_2\}$, where $Alph' \cap \{\surd, \surd_1, \surd_2\} = \emptyset$. Then for $S, S' \in Term_{GAP}$

$$S; S' := (Sg_1 \mid \bar{\surd}_1 S') - \{\surd_1, \bar{\surd}_1\}, \text{ where } g_1 = \{\surd \mapsto \surd_1, \bar{\surd} \to \bar{\surd}_1\}$$

Another very important abstract programming language based on Hoare's CSP ([Hoa78], see also [BRR87]) is TCSP; a slightly restricted variant can be defined, using the above, by the following grammar:

$$S ::= V \mid r \mid Sf \mid S\text{-}A \mid S \setminus A \mid S \text{ or } S \mid S;S \mid S \parallel_A S \mid rec\, r.S$$
$$V ::= nil \mid aS \mid V + V$$

where $r \in Id$, $a \in Alph$, $A \subseteq Alph$, $f \in Fun \wedge f\lceil_{(Act-Alph)} = id \wedge (Alph)f \subseteq Alph \wedge \forall a \in Alph : |af^{-1}| \in \mathbb{N}$.

The main omission is the operator [] of external choice, it is replaced by + which can be considered as [] restricted to operands which both begin by a visible action (according to the subgrammar with start symbol V). More on TCSP follows in part 2.

Examples:

(1) $rec\, r.((ar; b\surd nil) + \surd nil)$
An observer watching the system from a start action until a termination will note a sequence of actions of the form $a^n b^n \surd$, $n \geq 0$.

(2) $rec\, r.(0(rf) + \surd nil)$, where $af := \begin{cases} a+1 & \text{if } a \in \mathbb{N} \\ a & \text{otherwise} \end{cases}$, can produce each of the following action sequences:

$$\surd, 0\surd, 01\surd, 012\surd, 0123\surd, \ldots$$

(3) $rec\, r.(rf + 0\surd nil)$, where f is as above, produces only the actions $i\surd$ (where $i \in \mathbb{N}$) with increasing i (beginning with 0) if it is restarted again and again. For more details see [Tau88].

2 Semantics

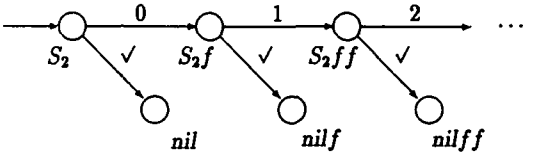
2.1 Interleaving Operational Semantics

The traditional approach to the semantics of concurrent distributed systems is based on the idea of an observer (or user) watching (or interacting with) the system **without** any knowledge about its structure. This observer (user) can only operate **sequentially**, so he will note (or cause) concurrent actions

of the system in some order - thus transforming concurrency into nondeterminism.

More formally, we associate to a term of the language (i.e. in our case GAP) a **transition system** (sequential automaton) T over $Act\,T = \langle Z, D, z \rangle$, where
Z is the (possibly infinite) set of states,
$D \subseteq Z \times Act \times Z$ is the set of transitions
and $z \in Z$ is the start state.

Example (2) from part 1 gives the following transition system: Let S_2 denote the given term



The states of the transition system for a term S are terms derived from S (where S is the start state) by the following inference rules – only the states reachable from the start state are interesting and need to be constructed in a concrete example.

$$(act) \qquad \frac{}{(cS, c, S)}$$

$$(fun) \qquad \frac{(S, a, S') \wedge af \neq \bot}{(Sf, af, S'f)}$$

$$(sum) \qquad \frac{(S, a, S')}{(S + R, a, S') \wedge (R + S, a, S')}$$

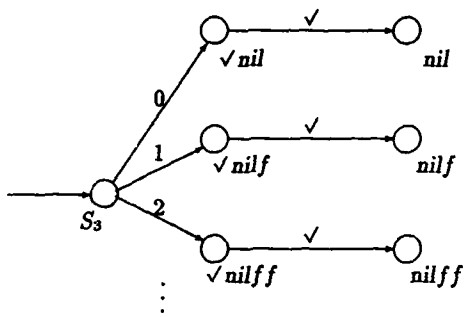
$$(asy) \qquad \frac{(S, a, S')}{(S \nmid R, a, S' \nmid R) \wedge (R \nmid S, a, R \nmid S')}$$

$$(syn) \qquad \frac{a, b \in Vis \wedge (S, a, S') \wedge (R, b, R')}{(S \nmid R, \{a, b\}, S' \nmid R')}$$

$$(rec) \qquad \frac{S \equiv rec\, r.R \wedge (R[S/r], a, S')}{(S, a, S')}$$

where $R[S/r]$ denotes the term obtained from R by substituting the term S for every free occurrence of the identifier r together with appropriate renaming of bound identifiers to avoid name clashings.

Example (3) from part 1 gives the transition system



If we consider such a transition system (for a term S) as an automaton whose final states are those reached by a tick transition $(z, \surd, z')$, then the formal language accepted by it, is the set of all sequences of observations (or of actions) one can obtain from terminating runs of an implementation of the term S.

The semantics obtained is an operational one constructed according to the structured-operational semantics (SOS) technique introduced by Plotkin; the semantics of the parallel composition of two terms is the interleaving (or the shuffle product, if formal languages are considered,) of the semantics (of the sets of action sequences) of the components.

Two terms have the same meaning with respect to this semantics if the corresponding transition systems are equivalent – to compare only the sets of action sequences does not suffice, since it does not say anything about the nonterminating behaviour. There are several equivalence notions for transition systems, we consider only the **strong bisimulation equivalence** (introduced by Milner and Park), since practically all other equivalence notions are weaker than this.

Let $T_i = \langle Z_i, D_i, z_i \rangle, i = 1, 2$ be two transition systems. T_1 and T_2 are strongly bisimular (notation: $T_1 \sim T_2$) if there is $B \subseteq Z_1 \times Z_2$ such that

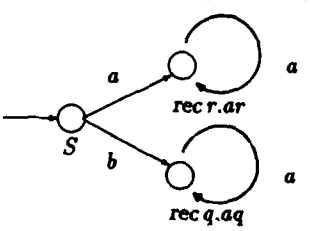
- $(z_1, z_2) \in B$
- $\forall (z, z') \in B, a \in Act$:

  (i) $(z, a, z_0) \in D_1 \Rightarrow \exists z_0' : (z', a, z_0') \in D_2 \wedge (z_0, z_0') \in B$

  (ii) $(z', a, z_0') \in D_2 \Rightarrow \exists z_0 : (z, a, z_0) \in D_1 \wedge (z_0, z_0') \in B$.

Obviously two terms with bisimular transition systems produce the same sets of action sequences.

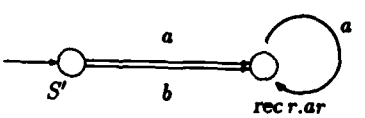
Example: $S \equiv a(\mathrm{rec}\, r.ar) + b(\mathrm{rec}\, q.aq)$ with $r \neq q$ has the transition system



According to the Barendregt convention (see part 1) we wanted to identify $\mathrm{rec}\, r.ar$ and $\mathrm{rec}\, q.aq$. Therefore S should be identified with

$$S' \equiv (a\mathrm{rec}\, r.ar + b\mathrm{rec}\, r.ar)$$

The transition system for S' is



Both transition systems are strongly bisimular. For further information on this semantics see [Tau88].

2.2 A Denotational Semantics with Simultaneity

We will now refine our semantical view. If the observer (or the user) is a bit more sophisticated he may detect (or cause) some actions simultaneously. It is obvious that for the analysis of a concurrent distributed system it is useful (and sometimes necessary) to be able to describe simultaneity of actions, e.g. in order to see what effect to execution speed the increase or decrease of a number of processors would have.

For this purpose we consider a TCSP oriented variant of GAP (called GAPH), take the (more or less) standard denotational semantics for TCSP (which however is an interleaving semantics) and equip this with the notion of step, which comes from Petri net theory.

A step is simply a finite multiset of actions which are performed simultaneously – it need not be maximal with respect to the number of simultaneous actions, since we assume that the components of the distributed system operate asynchronously. Also null steps (which do not contain any action) are allowed – they can be interpreted as idle steps but have nothing to do with τ-actions (which are not allowed in GAPH).

Since we allow for arbitrary simultaneity and are able to argue about simultaneity in our semantics it is useful to introduce a new operator ($\|_B$) in the language which allows to restrict simultaneity of certain actions (i.e. $S\|_B$ has the effect that no step which has an element of B as a substep can be performed).

The syntax of GAPH is given by the following grammar:

$$S ::= nil \mid r \mid aS \mid Sf \mid S\|_B \mid S \,[]\, S \mid S \text{ or } S \mid S \nmid S \mid \text{rec}\, r.S$$

where $r \in Id$, $a \in Alph$, $f \in Fun$, $B \subseteq \mathbf{M}$, where $\mathbf{M}$ is the set of all multisets over $\Sigma := Vis \cup EVis$; $\tilde{0}$ denotes the empty multiset.

Apart from $\|_B$ it is the operator [] of external choice (external nondeterminism) from TCSP, which offers a choice between two systems that is resolved by the environment (and which generalizes the + from GAP), that makes the difference to GAP.

The semantics is defined using the standard denotational technique: We first define a domain $\mathbf{F}$ (in our case a complete partial order) and then define for each syntactic operator *op* a corresponding continuous operator op_F on this domain. In order not to have to use environments we restrict our considerations to closed terms (i.e. terms not containing free identifiers).

The elements of the domain are sets $F \subseteq \mathbf{M}^* \times \mathbf{P}(\mathbf{M})$ of pairs consisting of a *sequence of steps* (the system may perform) and of a *set of steps* (the system may refuse to perform after having performed the before-mentioned steps).

Example: $S \equiv a\,nil \parallel_\emptyset (a\,nil \,[]\, b\,nil)$, $Vis = \{a, b, \surd\}$

S may perform the step $\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]$ (or $\left[\begin{smallmatrix} a \\ a \end{smallmatrix}\right]$) or may perform the sequence $[a][a]$ (or $[a][b]$ or $[b][a]$) of steps, and after that all steps (other than $\bar{0}$) can be refused; at the beginning only steps containing at least 2 simultaneous b's or at least 3 a's or one b and two a's or containing the $\surd$ can be refused; after one a or one b is performed in a single step only steps containing at least 2 simultaneous a's, one b or one $\surd$ can be refused. Therefore the step failure semantics of S is

$$\{\varepsilon X \mid X \subseteq \{x \in \mathbf{M} \mid x \geq \left[\begin{smallmatrix} b \\ b \end{smallmatrix}\right] \vee x \geq \left[\begin{smallmatrix} a \\ a \\ b \end{smallmatrix}\right] \vee x \geq \left[\begin{smallmatrix} a \\ a \\ a \end{smallmatrix}\right] \vee x \geq [\surd]\}\} \cup$$
$$\{[a]X, [b]X \mid X \subseteq \{x \in \mathbf{M} \mid x \geq \left[\begin{smallmatrix} a \\ a \end{smallmatrix}\right] \vee x \geq [b] \vee x \geq [\surd]\}\} \cup$$
$$\{\left[\begin{smallmatrix} a \\ a \end{smallmatrix}\right] X, \left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right] X, [a][a]X, [a][b]X, [b][a]X \mid X \subseteq \mathbf{M}\text{-}\{\bar{0}\}\}$$

For the description of the domain we need the concept of *stretching* of a step sequence, i.e. of replacing the step sequence by a step sequence performing the same individual actions in more (and smaller) steps; i.e. stretching means partial sequentialization plus insertion of null steps. Thus we can define the mapping $Stretch(w)$ inductively by:

$$\begin{aligned} Stretch(\varepsilon) &:= \{\bar{0}\}^* \\ Stretch(vx) &:= Stretch(v)\{x_1 x_2 \ldots x_n \in \mathbf{M}^* \mid \sum_{i=1}^{n} x_i = x\}. \end{aligned}$$

Definition: $F \subseteq \mathbf{M}^* \times \mathbf{P}(\mathbf{M})$ is an element of $\mathbf{F}$ iff

(1) $\varepsilon\emptyset \in F$

(2) $vw\emptyset \in F \Rightarrow v\emptyset \in F$

(3) $wX \in F \wedge Y \subseteq X \Rightarrow wY \in F$

(4) $wX \in F \wedge wy\emptyset \notin F \Rightarrow w(X \cup \{y\}) \in F$

(5) $(\forall Y \in \mathbf{P}(X) : wY \in F) \Rightarrow wX \in F$

(6) $v\{\bar{0}\} \in F \Rightarrow vwX \in F$

(7) $wX \in F \wedge v \in \mathit{Stretch}(w) \Rightarrow vX \in F$

(8) $wX \in F \wedge x \in X \wedge x \leq y \Rightarrow w(X \cup \{y\}) \in F$

(9) $v\bar{0}wX \in F \Leftrightarrow vwX \in F$

(10) $(\overset{\infty}{\exists} a \in \Sigma : w[a]\emptyset \in F) \Rightarrow w\{\bar{0}\} \in F$

It can be shown (see [TV]) that $(\mathbf{F}, \supseteq)$ is a cpo with bottom element $\bot = \mathbf{M}^* \times \mathbf{P}(\mathbf{M})$.

Now we can define the operators op_F which we denote by the same symbols as in the syntax.

$$
\begin{array}{lcl}
\mathit{nil} & := & \{wX \mid w \in \{\bar{0}\}^* \wedge X \subseteq \mathbf{M}\text{-}\{\bar{0}\}\} \\
aF & := & \{vX \mid v \in \{\bar{0}\}^* \wedge [a] \notin X \subseteq \mathbf{M}\text{-}\{\bar{0}\}\} \cup \\
 & & \{v[a]wX \mid v \in \{\bar{0}\}^* \wedge wX \in F\} \\
F_1 \textbf{ or } F_2 & := & F_1 \cup F_2 \\
F_1 \,[]\, F_2 & := & \{wX \mid w \in \{\bar{0}\}^* \wedge wX \in F_1 \cap F_2\} \cup \\
 & & \{wX \mid w \in \{\bar{0}\}^* \wedge \varepsilon\{\bar{0}\} \in F_1 \cup F_2 \wedge X \subseteq \mathbf{M}\} \cup \\
 & & \{wX \mid w \notin \{\bar{0}\}^* \wedge wX \in F_1 \cup F_2\} \\
F \between_B & := & \{w(X \cup Y) \mid w \in (\mathbf{M}\text{-}\overline{B})^* \wedge wX \in F \wedge Y \subseteq \overline{B}\} \cup \\
 & & \{wuX \mid (w\{\bar{0}\} \in F \vee \bar{0} \in B) \wedge w \in (\mathbf{M}\text{-}\overline{B})^* \wedge \\
 & & uX \in \mathbf{M}^* \times \mathbf{P}(\mathbf{M})\}, \\
 & & \text{where } B \subseteq \mathbf{M} \text{ and } \overline{B} := \{y \mid \exists x \in B : x \leq y\}
\end{array}
$$

To define $\between$ we need several auxiliary definitions. For each $t : \mathit{Vis} \times \mathit{Vis} \to \mathbf{N}$ we define the following three elements of $\mathbf{M}$ (considered as mappings)

$$\pi_1(t)(a) := \begin{cases} 0 & \text{if } a \in \mathit{EVis} \\ \sum_{b \in \mathit{Vis}} t(a,b) & \text{if } a \in \mathit{Vis} \end{cases}$$

$$\pi_2(t)(a) := \begin{cases} 0 & \text{if } a \in EVis \\ \sum_{b \in Vis} t(b,a) & \text{if } a \in Vis \end{cases}$$

$$\varphi(t)(a) := \begin{cases} 0 & \text{if } a \in Vis \\ t(b,b) & \text{if } a = \{b,b\} \in EVis \\ t(b,c) + t(c,b) & \text{if } a = \{b,c\}, \quad b \neq c \end{cases}$$

Let x_1, x_2 be steps, then $x_1 \nmid x_2$ is defined by

$$\begin{aligned} x \in x_1 \nmid x_2 \quad \text{iff} \quad & \exists r_1, r_2 \in \mathbf{M}, t : Vis \times Vis \to I\!N : \\ & x_1 = \pi_1(t) + r_1 \wedge x_2 = \pi_2(t) + r_2 \wedge \\ & x \in r_1 + r_2 + \varphi(t) \end{aligned}$$

Let $v = x_1 x_2 \ldots x_n$ and $w = y_1 y_2 \ldots y_n$ be step sequences of equal length, then

$$v \nmid w = \{z_1 z_2 \ldots z_n \mid z_i \in x_i \nmid y_i, \quad i = 1, \ldots, n\}$$

Now for $F_1, F_2 \in \mathbf{F}$ we have

$$\begin{aligned} F_1 \nmid F_2 := \{ & wX \mid \exists w_1 X_1 \in F_1, w_2 X_2 \in F_2 : w \in w_1 \nmid w_2 \wedge \\ & X \subseteq \{x \in \mathbf{M} - \{\bar{0}\} \mid \forall x_1, x_2 \in \mathbf{M} : x \in x_1 \nmid x_2 \\ & \qquad \Rightarrow x_1 \in X_1 \vee x_2 \in X_2\}\} \cup \\ & \{wuX \mid \exists w_1 X_1 \in F_1, w_2 X_2 \in F_2 : w = w_1 \nmid w_2 \wedge \\ & \qquad \wedge \bar{0} \in X_1 \cup X_2 \wedge u \in \mathbf{M}^* \wedge X \subseteq \mathbf{M}\}. \end{aligned}$$

Also, to define Sf for $f \in Fun$ we need several auxiliary definitions:

For $x \in \mathbf{M}$ and $f \in Fun$ we have:

xf undefined if $\exists a \in \Sigma : x(a) > 0 \wedge af = \perp$; in all other cases xf is the step (considered as a mapping from Σ to $I\!N$) defined by

$$(xf)(a) := \sum_{b \in \Sigma \wedge bf = a} x(b).$$

This implies in particular $\bar{0}f = \bar{0}$ for each $f \in Fun$.

Now

$$xf^{-1} := \{y \in \mathbf{M} \mid y_{\lceil (\tau f^{-1} \cap \Sigma)} = \bar{0} \wedge yf = x\}$$

and for $X \subseteq \mathbf{M}$

$$Xf^{-1} := \bigcup_{x \in X} xf^{-1}.$$

Let $w = x_1 x_2 \ldots x_n$ be a step sequence, then

$$wf := \begin{cases} \text{undefined} & \text{if one } w_i \text{ undefined} \\ w_1 w_2 \ldots w_n & \text{otherwise} \end{cases}$$

where

$$w_i := \begin{cases} \varepsilon & \text{if } x_i f = \bar{0} \neq x_i \\ x_i f & \text{otherwise} \end{cases}$$

Then for $F \in \mathbf{F}$ we get

$$\begin{aligned} Ff := \{ & wf(X \cup Y) \mid w(Xf^{-1} \cup \tau\widetilde{f^{-1}}) \in F \wedge \\ & Y \subseteq \{y \in \mathbf{M} \mid yf \text{ undef.}\}\} \cup \\ & \{(wf)uX \mid uX \in \mathbf{M}^* \times \mathbf{P}(\mathbf{M}) \wedge \\ & (w\{\bar{0}\} \in F \vee \forall n \in \mathbb{N} : \exists v \in (\tau\widetilde{f^{-1}})^n : wv\emptyset \in F)\} \end{aligned}$$

where for $\mathbf{A} \subseteq Act$ we write $\tilde{\mathbf{A}} := \{[a] \mid a \neq \tau\}$.

The proofs that all these operators are continuous can be found partly in [TV] – to prove that action manipulation (application of $f \in Fun$) is continuous, one cannot simply generalize the technique used in [TV] for the proof of the continuity of the hiding operator but has to go back to [Bro83].

The step failures semantics can be weakened to get new semantics by restricting the notion of steps and of refusals:

- if only null or singleton steps may be refused we get the simple step failures semantics
- if steps are restricted to be singletons only this gives the standard (linear) failures semantics.

Based on these notions of semantics we have different notions of equivalence.

Example:

$$\begin{aligned} S &:\equiv a\surd nil \parallel_{\{\surd\}} b\surd nil \\ T &:\equiv ab\surd nil \,[]\, ba\surd nil \end{aligned}$$

Then S and T have the same linear failures semantics but different (simple) step failures semantics and moreover
S, $S \,[]\, T$ and S or T have the same simple step failures semantics but
in the step failures semantics S and $S \,[]\, T$ have the same semantics while $S \,[]\, T$ and S or T have different semantics.

3 Petri Nets

3.1 Basic Ideas

Firstly we will consider (distributed concurrent) systems with the goal to develop a formalism that describes the structure as well as the dynamic behaviour of such systems. Using this formalism one can then specify new systems (which are to be built) also.

The basic assumptions from which we start are:

- Systems are composed of subsystems, which can communicate with each other (and with the environment) by sending and receiving messages (or other objects).
- The subsystems can be relatively independent of each other (e.g. they can be distributed widely).
- The behaviour of a system is determined by processes which are running in subsystems and which consist of changes of the states of subsystems by actions (of communication or transport).

The formal model is developed according to the following principles.

(1) States and actions (of state change) are both explicitely represented.

(2) States (resp. actions) of the subsystems are not combined together to form global states (resp. actions) of the whole system; they are represented separately.
Consequence: We better represent these systems by (at least) two-dimensional graphics.

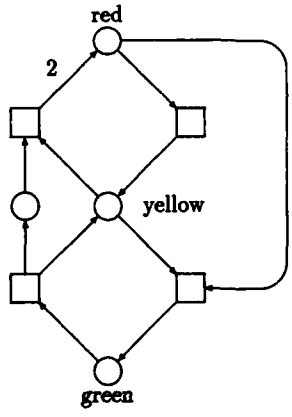
(3) The transport of an object (or a message) in the system can be considered as a state change. State changes can also be considered as actions of transport of objects (i.e. messages).
Consequences: For our formal description we need exactly 2 types of components and a notation for the objects and their position:

- active components (from a set T of so-called transitions, graphically represented by $\square$) for the representation of actions

- passive components (from a set S of so-called places graphically represented by ◯) for the representation of (local) states (of subsystems).
- As objects we consider (in the simplest version we want to discuss here) only simple tokens (graphically represented by a dot •) which are available in a place or not (⊙ or ◯).

(4) The amount of state change caused by one action is constant (i.e. always the same if the action occurs – independent of other circumstances)
Consequence: Each action component is connected to a fixed number of passive (state) components, which are subject to change by this action. Thus the system can be represented as a bipartite graph with node set $S \cup T$, where no two nodes of the same type are connected. Since state change is represented by taking away or adding tokens the graph will be directed (according to the direction of the flow of objects).

Example: Traffic light



(5) An action can take place (is enabled) if the state changes to be produced by it are possible; but the action need not take place if it is enabled such that by other actions it may be disabled again.

3.2 Formal Definitions

There are several ways to formally describe the class of Petri nets described above. The usual definition is the graph theoretic one, treating places and transitions equally:

(1) A **place/transition net** (P/T net) is a triple $N = (S, T, F)$, where S and T are the disjoint sets (of places and transitions) and F: $S \times T \cup T \times S \to \mathbb{N}$ (the flow relation).

Often one considers labelled P/T-nets, where different transitions may be equally labelled; i.e. one adds a labelling function l from T to a set of labels. If one wants to focus attention on the transitions (labelled by actions from a set Act of actions) then the following is more convenient. Let $\mathbf{M}(S)$ be the set of multisets over S.

(2) A P/T-net over Act is a pair (S, D) where S is the set of places, $D \subseteq \mathbf{M}(S) \times Act \times \mathbf{M}(S)$ is the set of labelled transitions (together with the weighted arrows connecting the related places) (For this notation see [Gol88]).

For more algebraic considerations a third definition seems to be promising (see [DMM89]).

(3) Let $S^{\oplus}$ be the free commutative monoid on S (if S is finite, the elements of $S^{\oplus}$ are the multisets over S), then an mP/T-net is a quadruple $N = (S^{\oplus}, T, \sigma, \Theta)$, where T is the set of transitions and and $\sigma, \Theta : T \to S^{\oplus}$ are mappings associating to every transition its pre-multiset and its post-multiset (places with the weighted arrows connecting them to the transition).

In the following we only consider P/T-nets where all weights of arrows are 1, i.e. where (in def. (1)) F maps into $\{0, 1\}$, i.e. where instead of $\mathbf{M}(S)$ we can use $\mathbf{P}(S)$ in definition (2) – moreover we will use definition (2). And we briefly call these particular P/T-nets only **nets**.

A **marking** of a net is a mapping $M : S \to \mathbb{N}$. A transition (S_1, a, S_2) is **enabled** at a marking M, if $S_1 \leq M$.

A transition d, enabled at a marking M may occur; if it occurs it produces the marking $M_2 := (M_1 - S_1) + S_2$; this is usually denoted by $M_1[d\rangle M_2$.

This notation is extended to arbitrary words over D by $M_1[\varepsilon\rangle M_1$ and $M_1[ud\rangle M_3$ iff $\exists M_2 : M_1[u\rangle M_2 \wedge M_2[d\rangle M_3$. Moreover we write $[M\rangle$ for the set of all markings reachable from M.

A net (S, D) with a marking M, called a marked net, is denoted as triple $(S, D; M)$.

3.3 Syntax-Driven Construction of Nets from GAP Terms

If we consider GAP as a specification language for concurrent systems and nets as formal descriptions of concurrent systems, it is natural to ask whether GAP terms can be represented as nets. Obviously the actions of GAP have to be represented by transitions, all the actions which may be performed before any other action have to be enabled by an appropriate marking; to the operators on terms should correspond operators on nets; i.e. we look for a syntax-driven modular net construction. Moreover we aim at "minimal" markings, i.e. mappings $S \to \{0, 1\}$, which we represent as subsets of S. In a first step we do not consider recursion. We again denote the operators in the same way as in GAP.

$$nil \quad := \quad (\{s\}, \emptyset; \{s\})$$

Let $N = (S, D; Z)$ be a marked net, then

$$aN \quad := \quad (S \cup \{s\}, D \cup \{(\{s\}, a, Z)\}; \{s\})$$
for $a \in Vis \cup \{\tau\}$.

Nf := *Reach*$(S, D'; Z)$, i.e.
the subnet reachable from the initially enabled transitions of the net $(S, D'; Z)$, where $D' = \{(M_1, af, M_2) \mid (M_1, a, M_2) \in D \wedge af \neq \perp\}$

To define the operator $+$ we introduce a restriction on the syntax: we allow only nets with a single marked place as operands (corresponding to GAP terms of the form aP) – otherwise problems would arise.

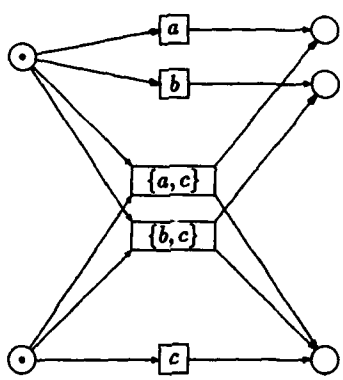
Let $N_i = (S_i, D_i; \{z_i\})$, $i = 1, 2$ be marked nets with $S_1 \cap S_2 = \emptyset$, then:

$$N_1 + N_2 \;:=\; Reach(S_1 \cup S_2 \cup \{z\}, D_1 \cup D_2 \cup D_+; \{z\}),$$
where $z \notin S_1 \cup S_2$ and
$$D_+ = \{(\{z\}, a, M) \mid (\{z_1\}, a, M) \in D_1 \vee (\{z_2\}, a, M) \in D_2\}$$

Let $N_i = (S_i, D_i; Z_i)$, $i = 1, 2$ be arbitrary marked nets with $S_1 \cap S_2 = \emptyset$, then:

$$N_1 \nmid N_2 \;:=\; (S_1 \cup S_2, D_1 \cup D_2 \cup D_+; Z_1 \cup Z_2), \text{ where}$$
$$D_+ = \{(M_1 \cup M_2, \{a_1, a_2\}, M_1' \cup M_2') \mid$$
$$\forall i \in \{1,2\} : a_i \in Vis \wedge (M_i, a_i, M_i') \in D_i\}$$

Example: Let $P :\equiv a\,nil + b\,nil$, $Q :\equiv c\,nil$
Then $P \nmid Q$ is represented by the net



The main problem is the modelling of recursion. Therefore we make another syntactical restriction: We consider only recursion terms of the form $rec\,p.\tau Q$. It can however be shown (see [Tau88]), that semantically this is not restrictive. An important goal of the construction is to obtain finite nets in as many cases as possible. The grammar for the terms to which we associate Petri nets now is as follows

$$S \;::=\; nil \mid p \mid aQ \mid S + S \mid rec\,p.\tau Q$$
$$Q \;::=\; S \mid Qf \mid Q \nmid Q.$$

The key idea for modelling recursion is based on Milner's construction of a finite extended transition for a CCS term (for details see [Tau88]); i.e. we enlarge the notion of a net by a means for representing identifiers and action manipulation functions: An **extended net** is a quadruple $N = (S, D, E; Z)$ where $(S, D; Z)$ is a net and

$$E \subseteq \mathbf{P}(S) \times Idf \times Fun_{\perp} \quad \text{(the set of extensions)}$$

($Fun_{\perp}$ is the set of action manipulation functions Fun, enlarged by the special element $\perp$, $\perp \notin Fun$)

An extension can be considered (and depicted) as a special transition with no post-set.

For $p \in Idf$ we then get the net representation

$$p := (\{z\}, \emptyset, \{(\{z\}, p, id)\}; \{z\})$$

All the other net constructions above must now be enlarged by appropriate extension sets E'

$$\begin{array}{rll}
nil: & E' := & \emptyset \\
Nf: & E' := & \{(M, p, gf) \mid (M, p, g) \in E \wedge g \neq \perp\} \cup \\
 & & \{(M, p, \perp) \mid (M, p, \perp) \in E\} \\
N_1 + N_2: & E' := & E_1 \cup E_2 \cup E_+ \text{ where} \\
 & E_+ := & \{(\{z\}, p, f) \mid (\{z_1\}, p, f) \in E_1 \vee (\{z_2\}, p, f) \in E_2\}. \\
N_1 \nmid N_2: & E' := & \{(M_1 \cup M_2, p, f) \mid (M_1, p, g) \in E_1 \wedge M_2 \in [Z_2)_2 \\
 & & \wedge (f = g \vee f = \perp \wedge \exists (U, a, U') \in D_2 \cup E_2 : U \subseteq M_2) \vee \\
 & & (M_2, p, g) \in E_2 \wedge M_1 \in [Z_1)_1 \\
 & & \wedge (f = g \vee f = \perp \wedge \exists (U, a, U') \in D_1 \cup E_1 : U \subseteq M_1)\}
\end{array}$$

To define the recursion operator for extended nets we need some more notations:

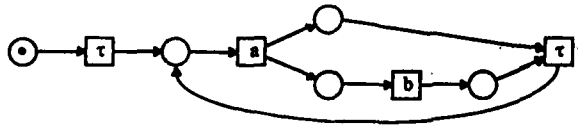
For $r \in Idf$ and $N = (S, D, E; Z)$ where
$\perp \notin E(r) := \{f \in Fun_{\perp} \mid (M, r, f) \in E\}$ define
$F := \{id \cdot f_1 \cdot \cdots \cdot f_n \mid n \geq 0, f_1, \ldots, f_n \in E(r)\}$, and
for $f \in F$ let $(S_f, D_f, E_f; Z_f) := Nf$ such that $f \neq g$ implies $S_f \cap S_g = \emptyset$.
Then

$$\begin{array}{rll}
\mathrm{rec}\, r.\tau N & := & Reach((\{z\} \cup \bigcup_{f \in F} S_f, \quad \{(\{z\}, \tau, Z_{id})\} \cup D_+ \cup \bigcup_{f \in F} D_f, \\
 & & \bigcup_{f \in F} E_f - (\mathbf{P}(\bigcup_{f \in F} S_f) \times \{r\} \times Fun); \{z\}))
\end{array}$$

where $z \notin S_f$ for all $f \in F$ and

$$D_+ = \{(M, \tau, Z_g) \mid (M, r, g) \in \bigcup_{f \in F} E_f\}$$

Example: $rec\, p.\tau a(\, nil \mid bp)$ gives the net



4 Semantics of Petri Nets

The simplest operational semantics of a marked net $(S, D; M)$ is the subset of all words $w \in D^*$ which denote sequences of occurrences of transitions starting from the marking M; one can refine this notion by considering an additional marking (or a set of markings) and take only those words in D^* which lead from M to this (these) marking(s). Thus a Petri net can be seen as a device to produce formal languages – and, indeed, there is a large body of interesting results on the formal languages of Petri nets; for details see the paper by M. Jantzen in [BRR87].

As Petri nets are meant to model not only relational systems (which accept input, produce output and stop) but also **reactive** systems (which are running all the time and react to interactions by users), it is useful to study also the infinite behaviour of nets – the simplest way to do it, is to study infinite sequences of transition occurrence possible in a net (see the papers by Carstensen in [CJK88] and by Valk in [BRR87]).

Since Petri nets should describe concurrent systems, it is also sensible to look after semantics that model concurrency more explicitely. Obviously the notion of steps (see part 2) can be used. A **step** of a net (S, D) is a multiset

over D. The step x is **enabled** at a marking M if for each $s \in S$

$$\sum_{d \in D} x(d) \cdot pr_1(d)(s) \leq M(s)$$

where $pr_1(d) = \delta_1$ if $d = (\delta_1, a, \delta_2) \in \mathbf{M}(S) \times Act \times \mathbf{M}(S)$.

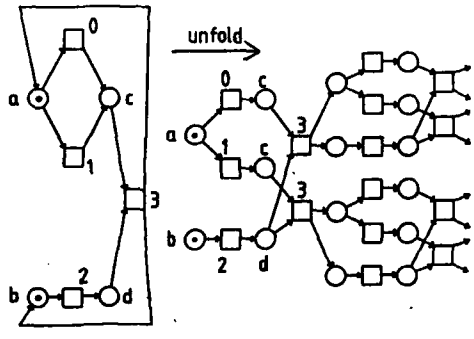
Now we can define finite (and infinite) step occurrence sequences in analogy to transition occurrence sequences.

Obviously we can also transfer the notion of a failures set to Petri nets (see e.g. the paper by Vogler in [CJK88]).

But Petri nets offer other interesting formalisms for describing the concurrent processes going on in a distributed system: One can represent these processes again as nets.

One possibility is to unfold a net (like one can unfold a while-loop into a sequence or a transition system into a tree).

Example: (from the paper by Winskel in [BRR87])



If one is only interested in the actions (the transitions) and the dependencies between the events of transition occurrences then one can erase the places in the unfolded net; one then obtain a so-called event structure – for more details on this very powerful semantical structure for describing concurrent processes see the paper by Winskel in [BRR87].

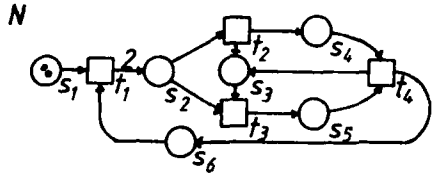
Instead of representing all possible processes in one unfolded net, one can separate them: beginning with an initial marking only that part of the unfolding is kept where the tokens flow through during a process. Then one marked net will be represented by a possibly infinite set of possibly infinite nets in which each place is in the preset of at most one transition only. For more details on the theory of these **nonsequential processes** see the paper by Fernandez in [BRR87] and the book [BF88].

If one suppresses the places in the nonsequential process, then one gets partial orders labelled with (the names of) transitions – these labelled partial orders can be considered as generalizations of words over the set of labels (words being linear labelled partial orders); therefore they are often called partial words. From the language of partial words defined by a Petri net one can easily obtain the language of transition sequences as well as the language of step sequences. For more details see [Kie88] and the paper by Kiehn in [Roz88].

There is one other quite interesting idea to give a non-interleaving (i.e. concurrency and nondeterminism distinguishing) semantics for (non-labelled) Petri nets. Let us imagine an observer (as in 2.1). We now ask: what minimal global, structural information on the net we need in order to infer from one observation sequence all other sequences possible by starting at the same marking. The answer (given by Marzurkiewicz), see his paper in [BRR87]) is: We only need to know which pairs of transitions are (always) independent such that they can be permuted in each transition sequence.

The original definition of independency of transitions has be generalized by Diekert (see [Die89]); it can however be further generalized as follows: Transitions t, t' are dependent iff $t = t'$ or $pr_3(t') \cap pr_1(t) \neq \emptyset$ or $pr_1(t') \cap pr_3(t) \neq \emptyset$. It is however not clear, what properties this generalized theory will have, since the extension of Marzurkiewicz's idea from the class of marked nets where each place can hold at most one token to general marked P/T-nets has some inherent difficulties lying in the fact that in the general case the permutability of transition occurrences in an observed sequence depends on the marking and not only on the structure of the net.

Example:



The main advantage of this approach is the following: The independency relation I on the set T of transitions can be used to define the free partially commutative monoid T^*/I of so-called traces (i.e. congruence classes of words with respect to permutation of independent transitions); the semantics of a net then becomes a subset of such a monoid; these monoids can be studied by algebraic techniques (see [Die89]).

5 Modular Construction and Refinement of Concurrent Systems

Abstract programming languages are based on the idea of modular construction; their semantics are always compositional.

Petri nets per se have no modular structure; only rather recently – based on the ideas from the field of abstract programming languages – modular construction techniques based on operators on nets have been studied (see [Tau88], [Gol88]). Also, it has been observed by Mazurkiewicz (see his paper in [Roz88]) that each unlabelled P/T-net (with arrow weights 1) can be considered as composed of so-called atomic nets (whose sets of places contain only one element) by an operation of synchronization (i.e. composition of nets by building the disjoint union of their places but perhaps identifying some of their transitions), yielding a compositional semantics for nets based on partial orders.

Another way to build complex structures is to use refinement techniques. Abstract programming languages pose problems with respect to refinement; e.g. bisimulation is not a congruence with respect to refinement.

Example: (see [GG89]): The two terms $P :\equiv a\,nil \parallel_\emptyset b\,nil$ and $Q :\equiv ab\,nil + ba\,nil$ are bisimulation equivalent but if the action a is refined into the sequential composition of two actions a_1, a_2 then we obtain two systems which are not bisimular:

$$P' :\equiv (a_1 a_2\,nil) \parallel_\emptyset b\,nil\ ,\ Q' :\equiv a_1 a_2 b\,nil + b a_1 a_2\,nil$$

Only recently the problem of refinement for abstract programming languages is being studied (see [GG89]).

For Petri nets refinement has been considered from the beginning on (see [Vog89]) – there are even several methods of refinement: One can refine single transitions or single places or one may cut off a part of a net (such that the boundary along the cut consists only of transitions or only of places) and substitute a new net into the remaining net. There are two different approaches to study this: Usually refinement is studied under the aspect of preserving the behaviour of the original net; however, recently (influenced by the abstract programming languages) also the situation is studied where the behaviour is changed by the refinement, such that the same refinements in two behaviourally equivalent nets result in behaviourally equivalent refined nets.

A completely different approach to refinement has been developed by A. Kiehn (see [Kie89]): Instead of replacing a transition by a net, an incarnation of the refining net is called (like a subroutine). This naturally also allows for recursion. Using this technique a concurrent system is specified by a set of nets which may call each other recursively. For the implementation of such specifications one obviously need some sort of stack mechanism.

Acknowledgement: I would like to thank R. Gold and D. Taubner for many helpful suggestions; in particular D. Taubner also extended the step failure semantics to the language GAPH. Special thanks go to H. Hadwiger and D. Stein for their excellent and phantastically quick typing of this text.

References

[BF88] E. Best and C. Fernandez. *Nonsequential Processes.* Number 13 in EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1988.

[Bra84] W. Brauer. How to play the token game. *Petri Net Newsletter*, 16:3–13, 1984.

[Bra87] W. Brauer. Carl Adam Petri and informatics. In G. Rozenberg K. Voss, H. Genrich, editor, *Concurrency and Nets*, pages 13–21. Springer-Verlag Berlin, Heidelberg, New York, Tokyo, 1987.

[Bro83] S. D. Brookes. *A Model for Communicating Sequential Processes.* Ph D thesis, Carnegie-Mellon Univ., 1983. Rpt. CMU-CS-83-149.

[BRR87] W. Brauer, W. Reisig, and G. Rozenberg, editors. *Petri Nets, Advances in Petri Nets 1986, Parts I and II*, LNCS 254 and 255. Springer-Verlag Berlin, Heidelberg, New York, Tokyo, 1987.

[CJK88] M. P. Chytil, L. Janiga, and V. Koubek, editors. *Mathematical Foundations of Computer Science 1988*, number 324 in Proc. MFCS, LNCS. Springer-Verlag Berlin, Heidelberg, New York, Tokyo, 1988.

[Die89] V. Diekert. Combinatorics on traces with applications to Petri nets and replacement systems. Habilitationsschrift, TU München, 1989. to be published in the LNCS.

[DMM89] P. Degano, J. Mesegner, and U. Montanari. Axiomatizing net computations and processes. In *Proc. 4th Ann. Symp. on Logic in Computer Science (LICS)*, Asilomar, Ca., USA, June 5-8 1989.

[GG89] R. van Glabbeck and U. Goltz. Partial order semantics for refinement of actions – neither necessary nor always sufficient but appropriate when used with care. In *EATCS Bull. No. 38*, pages 154–163, June 1989.

[Gol88] U. Goltz. Über die Darstellung von CCS-Programmen durch Petrinetze. *GMD-Bericht*, Nr. 172, Oldenburg-Verlag, München, Wien 1988. see also [CJK88], pages 339-350.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Comm. ACM*, 21:666–677, 1978.

[Kie88] A. Kiehn. On the interelation between synchronized and nonsynchronized behaviour of Petri nets. *EIK*, 24:3–18, 1988.

[Kie89] A. Kiehn. A structuring mechanism for Petri nets. Report TUM-I 8902, TU München, 1989.

[Mil85] R. Milner. Lectures on a calculus of communicating systems. In S. D. Brookes et. al., editor, *Seminar on Concurrency*, number 197 in LNCS, pages 197–220. Springer-Verlag Berlin, Heidelberg, New York, Tokyo, 1985.

[Roz88] G. Rozenberg, editor. *Advances in Petri Nets 1988.* Number 340 in LNCS. Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1988.

[Tau88] D. Taubner. *The Finite Representation of Abstract Programs by Automata and Petri Nets.* Dissertation, Tech. Univ. Munich, Report TUM-I 8817, Dec 1988. to appear in revised form in the LNCS.

[Tra88] B. A. Trakhtenbrot. Comparing the church and the turing approaches: Two prophetical messages. In R. Herken, editor, *The Universal Turing Machine, A Half-Century Survey*, pages 603–630. Kammerer & Unverzagt Hamburg, Berlin and Oxford University Press, 1988.

[TV] D. Taubner and W. Vogler. Step failure semantics and a complete proof system. to appear in Acta Informatica – preliminary versions appeared as report TUM-I 8614, Tech. Univ. Munich, Sept 1986 and in LNCS 247 (Proceedings STACS 87), pp. 348-359.

[Vog89] W. Vogler. Failures semantics of Petri nets and the refinement of places and transitions, manuscript. submitted to TCS, 1989.